Parallel Program Graphs and their Classification

Vivek Sarkar Barbara Simons

IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, CA 95141
({vivek_sarkar,simons}@vnet.ibm.com)

Abstract. We categorize and compare different representations of program dependence graphs, including the Control Flow Graph (CFG) which is a sequential representation lacking data dependences, the Program Dependence Graph (PDG) which is a parallel representation of a sequential program and is comprised of control and data dependences, and more generally, the Parallel Program Graph (PPG) which is a parallel representation of sequential and (inherently) parallel programs and is comprised of parallel control flow edges and synchronization edges. PPGs are classified according to their graphical structure and properties related to deadlock detection and serializability.

1 Introduction

The importance of optimizing compilers has increased dramatically with the development of multiprocessor computers. Writing correct and efficient parallel code is quite difficult at best; writing such software and making it portable currently is almost impossible. Intermediate forms are required both for optimization and portability. A natural intermediate form is the graph that represents the instructions of the program and the dependences that exist between them. This is the role played by the Parallel Program Graph (PPG). The Control Flow Graph (CFG), in turn, is required once portions of the PPG have been mapped onto individual processors. PPGs are a generalization of Program Dependence Graphs (PDGs), which have been shown to be useful for solving a variety of problems, including optimization [15], vectorization [7], code generation for VLIW machines [19], merging versions of programs [22], and automatic detection and management of parallelism [3, 33, 32]. PPGs are useful for determining the semantic equivalence of parallel programs [35, 10, 34], and also have applications to automatic code generation [25].

In section 2, we define PPGs as a more general intermediate representation of parallel programs than PDGs. PPGs contain *control* edges that represent parallel flow of control, and *synchronization* edges that impose ordering constraints on execution instances of PPG nodes. *MGOTO* nodes are used to create new threads of parallel control. Section 3 contains a few examples of PPGs to provide some intuition on how PPGs can be built in a compiler and how they execute at run-time. Section 4 characterizes PPGs based on the nature of their control edges and synchronization edges, and summarizes the classifications of PPGs according to the results reported in sections 5 and 6.

Unlike PDGs which can only represent the parallelism in a sequential program, PPGs can represent the parallelism in inherently parallel programs. Two important issues that are specific to PPGs are *deadlock* and *serializability*. A PPG's execution may deadlock if its synchronization edges are incorrectly specified i.e. if there is an execution sequence of PPG nodes containing a cycle. This is also one reason why PPGs are more general than PDGs. In section 5, we characterize the complexity of deadlock detection for different classes of PPGs. Serialization is the process of transforming a PPG into a semantically equivalent sequential CFG with neither node duplication nor creation of Boolean guards. It has already been shown that there exist PPGs that are not serializable [14, 38], which is another reason why PPGs are more general than PDGs. In section 6, we further study the serialization problem by characterizing the serializability of different classes of PPGs.

Finally, section 7 discusses related work, and section 8 contains the conclusions of this paper and an outline of possible directions for future work.

2 Definition of PPGs

2.1 Control Flow Graphs (CFGs)

We begin with the definition of a *control flow graph* as presented in [31, 11, 33]. It differs from the usual definition [1, 15] by the inclusion of a set of labels for edges and a mapping of node types. The label set $\{T, F, U\}$ is used to distinguish between conditional execution (labels T and F) and unconditional execution (label U).

Definition 1. A *control flow graph* $CFG = (N, E_{cf}, TYPE)$ is a rooted directed multigraph in which every node is reachable from the root. It consists of:

- N, a set of nodes. A CFG node represents an arbitrary sequential computation e.g. a basic block, a statement or an operation.
- $E_{cf} \subseteq N \times N \times \{T, F, U\}$, a set of *labelled* control flow edges.
- *TYPE*, a node type mapping. $TYPE(n)$ identifies the type of node n as one of the following values: *START*, *STOP*, *PREDICATE*, *COMPUTE*.

We assume that *CFG* contains two distinguished nodes of type *START* and *STOP* respectively, and that for any node N in *CFG*, there exist directed paths from *START* to N and from N to *STOP*.

Each node in a *CFG* has at most two outgoing edges. A node with exactly two outgoing edges represents a conditional branch (predicate); in that case, the node has $TYPE = PREDICATE$ and we assume that the outgoing edges are distinguished by "T" *(true)* and "F" *(false)* labels[1]. If a node has $TYPE = COMPUTE$ then it has exactly one outgoing edge with label "U" *(unconditional)*.

[1] The restriction to at most two outgoing edges is made to simplify the discussion. All the results presented in this paper are applicable to control flow graphs with arbitrary out-degree e.g. when representing a computed `GOTO` statement in Fortran, or a `switch` statement in C. Note that *CFG* imposes no restriction on the in-degree of a node.

□

A CFG is a standard sequential representation of a program with no parallel constructs. A CFG is also the target for code generation from parallel to sequential code.

2.2 Program Dependence Graphs (PDGs)

The *Program Dependence Graph* (PDG) [15] is a standard representation of control and data dependences derived from a sequential program and its CFG. Like CFG nodes, a PDG node represents an arbitrary sequential computation e.g. a basic block, a statement, or an operation. An edge in a PDG represents a *control dependence* or a *data dependence*. PDGs reveal the inherent parallelism in a program written in a sequential programming language by removing artificial sequencing constraints from the program text.

Definition 2. A *Program Dependence Graph* $PDG = (N, E_{cd}, E_{dd}, TYPE)$ is a rooted directed multigraph in which every node is reachable from the root. It consists of [15]:

- N, a set of nodes.
- $E_{cd} \subseteq N \times N \times \{T, F, U\}$, a set of labelled *control dependence* edges.
  Edge $(a, b, L) \in E_{cd}$ identifies a control dependence from node a to node b with label L. This means that a must have $TYPE = PREDICATE$ or $TYPE = REGION$. If $TYPE = PREDICATE$ and if a's predicate value is evaluated to be L, then it causes an execution instance of b to be created. If $TYPE = REGION$, then an execution instance of b is always created [15, 33].
- $E_{dd} \subseteq N \times N \times Contexts$, a set of *data dependence* edges.
  Edge $(a, b, C) \in E_{dd}$ identifies a data dependence from node a to node b with context C. The context C identifies the pairs of execution instances of nodes a and b that must be synchronized. *Direction vectors* [40] and *distance vectors* [26] are common approaches to representing contexts of data dependences; the *gist* [30] and the *last-write tree* [27] are newer representations for data dependence contexts that provide more information. Context information can identify a data dependence as being *loop-independent* or *loop-carried* [4]. In the absence of any other information, the context of a data dependence edge in a PDG must at least be *plausible* [8] i.e. the execution instances participating in the data dependences must be consistent with the sequential ordering represented by the CFG from which the PDG is derived.
- $TYPE$, a node type mapping. $TYPE(n)$ identifies the type of node n as one of the following values: $START$, $PREDICATE$, $COMPUTE$, $REGION$.
  The $START$ node and $PREDICATE$ node types in a PDG are like the corresponding node types in a CFG. The $COMPUTE$ node type is slightly different because it has no outgoing control edges in a PDG. By contrast, every node type except $STOP$ is required to have at least one outgoing edge in a CFG. A node with $TYPE = REGION$ serves as a summary node for a

set of control dependence successors in a PDG. Region nodes are also useful for *factoring* sets of control dependence successors [15, 12].

□

2.3 Parallel Program Graphs (PPGs)

The *Parallel Program Graph* (PPG) is a general intermediate representation of parallel programs that includes PDGs and CFGs. Analogous to control dependence and data dependence edges in PDGs, PPGs contain *control* edges that represent parallel flow of control and *synchronization* edges that impose ordering constraints on execution instances of PPG nodes. PPGs also contain *MGOTO* nodes that are a generalization of *REGION* nodes in PDGs. What distinguishes PPGs from PDGs is the complete freedom in connecting PPG nodes with control and synchronization edges to span the full spectrum of sequential and parallel programs. Control edges in PPGs can be used to model program constructs that cannot be modelled by control dependence edges in PDGs e.g. PPGs can represent *non-serializable* parallel programs [37, 36, 34]. Synchronization edges in PPGs can be used to model program constructs that cannot be modelled with data dependence edges in PDGs e.g. a PPG may enforce a synchronization with distance vector (-1), from the "next" iteration of a loop to the "previous" iteration, as in (say) Haskell array constructors [24] or I-structures [5], while such data dependences are prohibited in PDGs because they are not plausible with reference to the sequential program from which the PDG was derived.

Definition 3. A *Parallel Program Graph* $PPG = (N, E_{cont}, E_{sync}, TYPE)$ is a rooted directed multigraph in which every node is reachable from the root using only *control* edges. It consists of [34]:

1. N, a set of nodes.
2. $E_{cont} \subseteq N \times N \times \{T, F, U\}$, a set of labelled *control* edges. Edge $(a, b, L) \in E_{cont}$ identifies a control edge from node a to node b with label L.
3. $E_{sync} \subseteq N \times N \times SynchronizationConditions$, a set of *synchronization* edges. Edge $(a, b, f) \in E_{sync}$ defines a synchronization from node a to node b with synchronization condition f.
4. *TYPE*, a node type mapping. $TYPE(n)$ identifies the type of node n as one of the following values: *START*, *PREDICATE*, *COMPUTE*, *MGOTO*.
   The *START* node and *PREDICATE* node types in a PPG are just like the corresponding node types in a CFG or a PDG. The *COMPUTE* node type is more general because a PPG compute node may either have an outgoing control edge as in a CFG or may have no outgoing control edges as in a PDG. A node with $TYPE = MGOTO$ is used as a construct for creating parallel threads of computation: a new thread is created for each successor of an *MGOTO* node. *MGOTO* nodes in PPGs are a generalization of *REGION* nodes in PDGs.

We distinguish between a PPG node and an *execution instance* of a PPG node (i.e., a dynamic instantiation of that node). Given an execution instance I_a of PPG node a, its *execution history*, $H(I_a)$, is defined as the sequence of PPG node and label values that caused execution instance I_a to occur. A PPG's execution begins with a single execution instance (I_{start}) of the *start* node, with $H(I_{start}) = <>$ (an empty sequence).

A *control* edge in E_{cont} is a triple of the form (a, b, L), which defines a transfer of control from node a to node b for branch label (or branch condition) L. The semantics of *control* edges is as follows. If $TYPE(a) = PREDICATE$, then consider an execution instance I_a of node a that evaluates node a's branch label to be L: if there exists an edge $(a, b, L) \in E_{cont}$, execution instance I_a creates a new execution instance I_b of node b (there can be at most one such successor node) and then terminates itself. The execution history of each I_b is simply $H(I_b) = H(I_a) \circ < a, L >$ (where $\circ$ is the sequence concatenation operator). If $TYPE(a) = MGOTO$, then let the set of outgoing control edges from node a with branch label L (which must be $= U$) be $\{(a, b_1, L), \ldots, (a, b_k, L)\}$. After completion, execution instance I_a creates a new execution instance (I_{b_i}) of each target node b_i and then terminates itself. The execution history of each I_{b_i} is simply $H(I_{b_i}) = H(I_a) \circ < a, L >$.

A *synchronization* edge in E_{sync} is a triple of the form (a, b, f), which defines a PPG edge from node a to node b with synchronization condition f. $f(H_1, H_2)$ is a computable Boolean function on execution histories. Given two execution instances I_a and I_b of nodes a and b, $f(H(I_a), H(I_b))$ returns *true* if and only if execution instance I_a must complete execution before execution instance I_b can be started. For theoretical reasons, it is useful to think of synchronization condition f as an arbitrary computable Boolean function; however, in practice, we will only be interested in simple definitions of f that can be stored in at most $O(|N|)$ space. Note that the synchronization condition depends only on execution histories, and not on any program data values. Also, due to the presence of synchronization edges, it is possible to construct PPGs that may deadlock. □

A PPG node represents an arbitrary sequential computation. The control edges of a PPG specify how execution instances of PPG nodes are created (unravelled), and the synchronization edges of a PPG specify how execution instances need to be synchronized. A formal definition of the execution semantics of *mgoto* edges and *synchronization* edges is given in [34].

We would like to have a definition for synchronization edges that corresponds to the notion of *loop-independent* data dependence edges in sequential programs [40]. If nodes a and x are in a loop in a sequential program and have a loop-independent data dependence between them, then whenever x is executed in an iteration of the loop, an instance of a also must be executed prior to x *in that same loop iteration.* However, if in some iteration of the loop x is not executed, say because of some conditional branch, then we don't care whether or not a is executed. This is the notion we are trying to capture with our definition of *control-independent*, below. First we need a couple of other definitions.

Consider an execution instance I_x of PPG node x with execution history $H(I_x) =< u_1, L_1, \ldots, u_i, \ldots, u_j, \ldots, u_k, L_k >$, $u_k = x$ (as defined in [34]). We define *nodeprefix*$(H(I_x), a)$ (the *nodeprefix* of execution history $H(I_x)$ with respect to the PPG node a) as follows. If there is no occurrence of node a in $H(I_x)$, then *nodeprefix*$(H(I_x), a) = \perp$. If $a \neq x$, *nodeprefix*$(H(I_x), a) =< u_1, L_1, \ldots, u_i, L_i >, u_i = a, u_j \neq a, i < j \leq k$; if $a = x$, then *nodeprefix*$(H(I_x), a)$ $= H(I_x) =< u_1, L_1, \ldots, u_i, \ldots, u_j, \ldots, u_k, L_k >$.

A *control path* in a PPG is a path that consists of only control edges. A node a is a *control ancestor* of node x if there exists an acyclic control path from $START$ to x such that a is on that path.

A synchronization edge (x, y, f) is *control-independent* if a necessary (but not necessarily sufficient) condition for $f(H(I_a), H(I_b)) = true$ is that *nodeprefix*$(H(I_x), a)$ $=$ *nodeprefix*$(H(I_y), a)$ for all nodes a that are control ancestors of both x and y.

3 Examples of PPGs

In this section, we give a few examples of PPGs to provide some intuition on how PPGs can be built in a compiler and how they execute at run-time.

Figure 1 is an example of a PPG obtained from a thread-based parallel programming model that uses `create` and `terminate` operations. This PPG contains only control edges; it has no synchronization edges. The `create` operation in statement S2 is modeled by an *MGOTO* node with two successors: node S6, which is the target of the `create` operation, and node S3, which is the continuation of the parent thread. The `terminate` operation in statement S5 is modeled by making node S4 a leaf node i.e. a node with no outgoing control edges.

Figure 2 is another example of a PPG that contains only control edges and is obtained from a thread-based parallel programming model. This is an example of "unstructured" parallelism because of the `goto S4` statements which cause node S4 to be executed twice when predicate S1 evaluates to true. The PPG in Figure 2 is also an example of a non-serializable PPG. Because node S4 can be executed twice, it is not possible to generate an equivalent sequential CFG for this PPG without duplicating code or inserting Boolean guards. With code duplication, this PPG can be made serializable by splitting node S4 into two copies, one for parent node S2 and one for parent node S3.

Figure 3 is an example of a PPG that represents a `parallel sections` construct [29], which is similar to the `cobegin-coend` construct [20]. This PPG contains both control edges and synchronization edges. The start of a `parallel sections` construct is modelled by an *MGOTO* node with three successors: node S1, which is the start of the first parallel section, node S2, which is the start of the second parallel section, and node S5, which is the continuation of the parent thread. The `goto S0` statement at the bottom of the code fragment is simply modelled by a control edge that branches back to node S0.

PARALLEL THREADS EXAMPLE

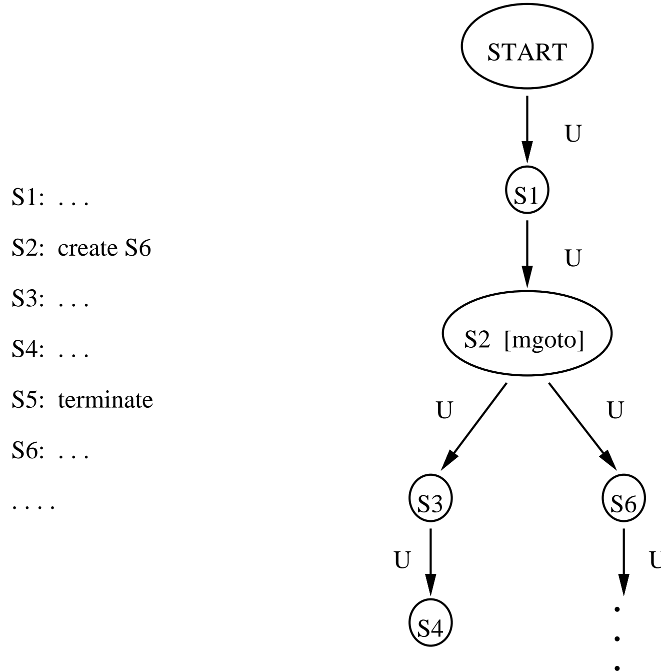

Fig. 1. Example of PPG with only control edges

The semantics of `parallel sections` requires a synchronization at the `end sections` statement. This is modelled by inserting a synchronization edge from node S2 to node S5 and another synchronization edge from node S4 to node S5. The synchronization condition f for the two synchronization edges is defined in Figure 3 as a simple function on the execution histories of execution instances I.S2, I.S4, I.S5 of nodes S2, S4, S5 respectively. For example, the synchronization condition for the edge from S2 to S5 is defined to be true if and only if $H(I.S2) = concat(H(I.S5), < S1, U >)$; this condition identifies the synchronization edge as being control-independent. In practice, a control-independent synchronization is implemented by simple semaphore operations without needing to examine the execution histories at run-time [18].

As an optimization, one may observe that the three-way branching at the $MGOTO$ node (S0) in Figure 3 can be replaced by two-way branching without any loss of parallelism. This is because node S5 has to wait for nodes S2 and S4 to complete before it can start execution. A more efficient PPG for this example can be obtained by deleting the control edge from node S0 to node S5, and then replacing the synchronization edge from node S2 to node S5

EXAMPLE OF UNSTRUCTURED PARALLELISM

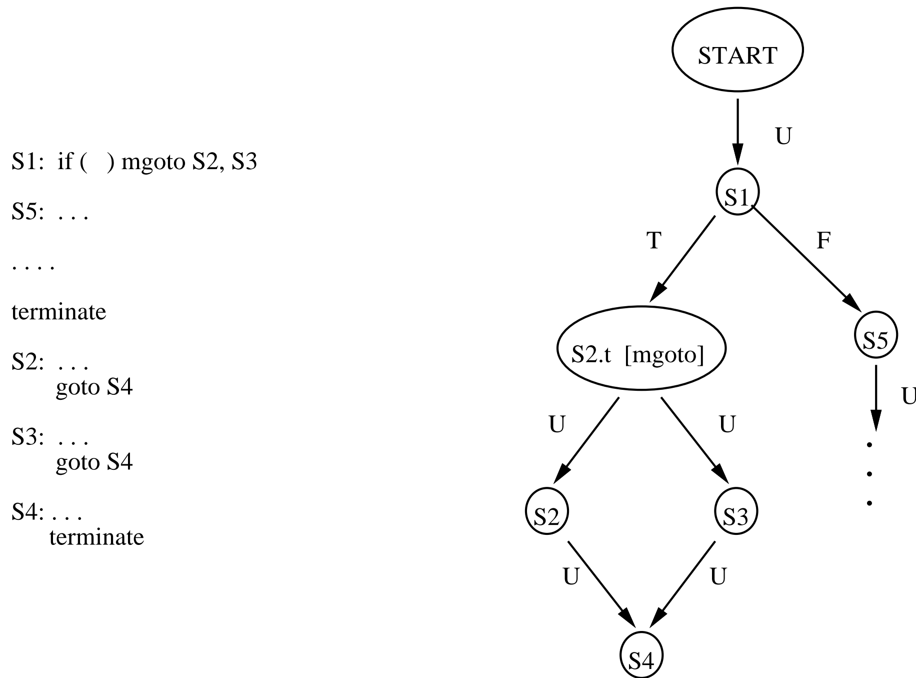

Fig. 2. Example of unstructured PPG with only control edges

by a control edge. This transformed PPG would contain the same amount of ideal parallelism as the original PPG but would create two threads instead of three, and would perform one synchronization instead of two, for a given instantiation of the `parallel sections` construct[2]. This idea is an extension of the *control sequencing* transformation used in the PTRAN system to replace a data synchronization by sequential control [13, 33].

Further, if the granularity of work being performed in the parallel sections is too little to justify creation of parallel threads, the PPG in Figure 3 can be transformed into a sequential CFG, which is a PPG with no $MGOTO$ nodes and no synchronization edges (this transformation is possible because the PPG in Figure 3 is serializable). Therefore, we see that PPG framework can can support a wide range of parallelism from ideal parallelism to useful parallelism to no

[2] Note that if both synchronization edges coming into S5 were replaced by control edges, the resulting PPG would perform node S5 twice for a given instantiation of the `parallel sections` construct, which is incorrect.

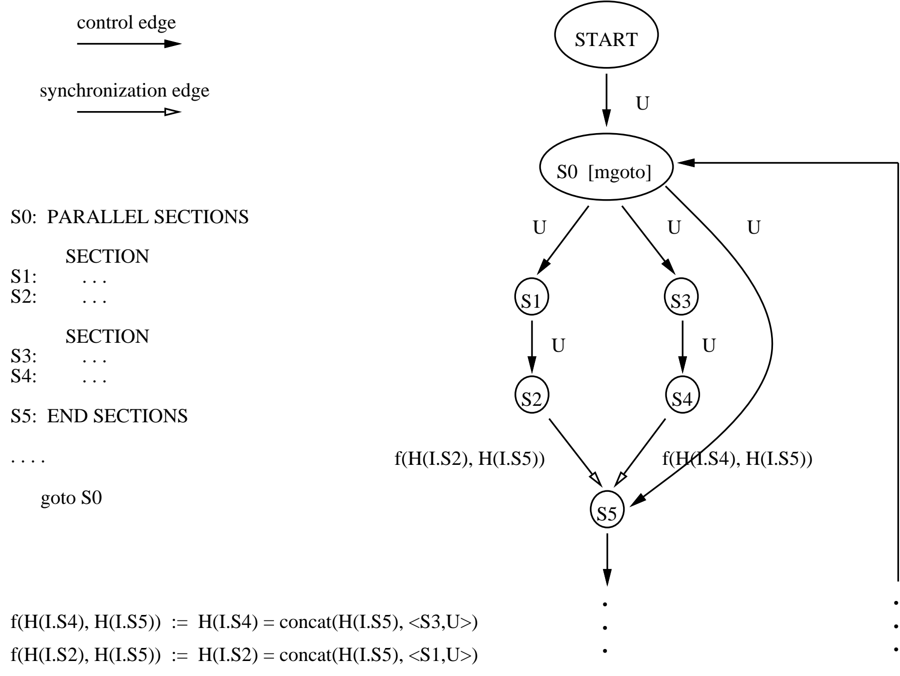

Fig. 3. Example of structured PPG with control and synchronization edges

parallelism. As future work, it would be interesting to extend existing algorithms for selecting useful parallelism from PDGs (e.g. [32]) to operate more generally on PPGs.

4 Classification of PPGs

4.1 Classification criteria

In this subsection, we characterize PPGs based on the nature of their control edges and synchronization edges.

Control edges. The control edges of PPGs can be *acyclic*, or they can have loops, which in turn can be *reducible* or *irreducible*. For reducibility, we extend the definition used for CFGs so that it is applicable to PPGs as well: a PPG is reducible if each strongly connected region of its control edges has a single entry node [2, 21, 1]; otherwise, a PPG is said to be irreducible. This classification of a PPG's control edges as acyclic/reducible/arbitrary is interesting because of the characterizations that can then be made. Note that each class is properly contained within the next, with irreducible PPGs being the most general case.

Synchronization edges. PPGs can also be classified by the nature of their synchronization edges as follows:

1. No synchronization edges:
   A CFG is a simple example of a PPG with no synchronization edges. Figures 1 and 2 in section 3 are some more examples of PPGs that have no synchronization edges.
2. control-independent synchronizations:
   In this case, all synchronization edges are assumed to have synchronization conditions that cause only execution instances from the same loop iteration to be synchronized[3]. For example, the two synchronizations in Figure 3 are control-independent.
3. General synchronizations:
   In general, the synchronization function can include control-independent synchronizations, loop-carried synchronizations, etc.

Predicate-ancestor condition.

Definition 4. Predicate-ancestor condition (pred-anc cond): If two control edges terminate in the same node, then the least common control ancestor (obtained by following only control edges) of the source nodes of both edges is a predicate node.

No-post-dominator condition.

Definition 5. No-post-dominator condition (no-post-dom cond): let P be a predicate and S a control descendant of P, then S does not post-dominate P if there is some path from P to a leaf node that excludes S. The no-post-dominator condition assumes that for every descendant S of predicate P, S does not post-dominate P.

4.2 Characterization of PPGs based on the Deadlock Detection Problem

Table 1 classifies PPGs according to how easy or hard it is to detect if a PPG in a given class can deadlock. A detailed discussion of these results is presented in section 5.

4.3 Characterization of PPGs based on Serializability

One of the primary distinctions between a CFG and a PDG or PPG is that the CFG is already serialized, while neither the PDG or the PPG is serialized. Table 2 classifies PPGs according to how easy or hard it is to detect if a PPG in a given class is serializable. A detailed discussion of these results is presented in Section 6. Note that serializability is a stronger condition than deadlock detection because a serializable program is guaranteed never to deadlock.

[3] We assume START and STOP are part of a dummy loop with one iteration.

Control edges	Sync. edges	Characterization
Arbitrary	None	PPGs in this class will never deadlock, because they have no synchronization edges (Lemma 6).
Acyclic	Acyclic (in conjunction with control edges)	PPGs in this class will never deadlock, because it is not possible for a cyclic wait to occur at run-time (Lemma 7).
Reducible	Control-independent, no synchronization back-edges	PPGs in this class will never deadlock (Lemma 8).
Acyclic, satisfies pred-anc and no-post-dominator conditions	control-independent	This is a special case of PPGs with acyclic control edges. There are known polynomial-time algorithms that can determine whether such a PPG is serializable. If it is serializable, then it is guaranteed to not deadlock.
Acyclic	control-independent	The problem of determining an assignment of truth values to the predicates that causes the PPG to deadlock is shown to be NP-hard. It is easy to specify an exponential-time algorithm for solving the problem by enumerating all possible assignments of truth values to predicates.
Arbitrary	Arbitrary	The problem of detetecting deadlock in arbitrary PPGs is a very hard problem. We conjecture that it is undecidable in general.

Table 1. Characterization of deadlock detection problem for PPGs

5 Deadlock detection

The introduction of synchronization edges makes deadlock possible. Deadlock occurs if there is a cyclic wait, so that every node in the cycle is waiting for one of the other nodes in the cycle to proceed. Formally, we say that a PPG *deadlocks* if there is an execution instance (dynamic instantiation) v_i of a node v, that instance is never executed (scheduled), no other new execution instances are created, and no other node is executed. We refer to v as a *deadlocking node* and v_i as a *deadlocking instance*; there might be multiple deadlocking nodes and instances. Note that our definition of deadlock implies that there is no infinite loop. The motivation for this definition is to distinguish between deadlock and non-termination.

Given a synchronization edge (u, v, f) and an execution instance v_i of v, we say that (u, v, f) is *irrelevant* for v_i if for all execution instances u_j of u that are created, $f(H(I_{u_j}), H(i_{v_i})) = false$, and for all u_j such that $f(H(I_{u_j}), H(i_{v_i})) = true$, no execution instance of u_j can ever be created. (u, v, f) is *relevant* for v_i if it is not irrelevant for v_i.

Control edges	Sync. edges	Characterization
Reducible, satisfy pred-anc and no-post-dominator conditions	control-independent	Section 6 outlines known algorithms for determining whether or not a PPG in this class is serializable (without node duplication or creation of Boolean guards).
Do not satisfy pred-anc condition	None	It is known that PPGs that do not satisfy the pred-anc condition are not serializable (without node duplication or creation of Boolean guards).

Table 2. Characterization of serializability detection of PPGs

Lemma 6. *A PPG with no synchronization edges will never deadlock.*

Proof: Follows from the definitions of control edges and deadlock. □

A PPG G is *acyclic* if it is acyclic when both the control and the synchronization edges are considered.

Lemma 7. *Let $G = (N, E)$ be an acyclic PPG. Then G never deadlocks.*

Proof: The proof is by induction on the size of N. If $|N| = 1$, then G consists of only *START* and will never deadlock. Suppose for induction that the lemma holds for all PPGs with k or fewer nodes, and consider $G = (N, E)$, with $|N| = k+1$. Suppose $v \in N$ is a deadlocking node with deadlocking instance v_i. By lemma 6 there must be a synchronization edge (u, v, f) that is relevant for v_i and that is causing v_i to deadlock. Since G is acyclic, $u \neq v$. By the definition of deadlock, v_i is created but not executed. Suppose v_i is the only instance of v that is created. If u or some predecessor of u is a deadlocking node, then we get a contradiction to the induction assumption by considering the subgraph of G that contains all nodes except for v. So either there are no instances of u created, or all instances u_j of u that are created are executed. Since (u, v, f) is relevant, either f will evaluate to *false* or (u_j, v_i, f) is not a deadlocking edge for all created u_j.

If multiple instances of v are created, then since there are no synchronization edges between any pair of instances of v, the instances are all independent. Consequently, the above argument holds for any deadlocking instance of v. □

A *cycle* is a path of directed synchronization and/or control edges such that the first and last nodes on the path are identical. Given cycle C, node $a \in C$ is an *entry point* of C if there exists some path π of all control edges from the root to a such that π contains no other nodes from C. A cycle is *single-entry* if it contains only one entry point; otherwise, it is *multiple-entry*.

Synchronization edge (v, u, f) or control edge (v, u) is a *back-edge* if u is a control ancestor of v. Note that the definition of back-edge can include self-loops, i.e. a synchronization edge (v, v, f).

Lemma 8. *Let $G = (N, E)$ be a PPG which contains only control-independent synchronization edges. Suppose also that there are no synchronization back-edges and that all cycles are single-entry. Let $G' = (N', E')$ be the PPG that is obtained from G by deleting all back-edges in G (i.e. $N' = N$ and $E' \subseteq E$). Then for all $a, x \in N$, if a is a control ancestor of x in G, then a is a control ancestor of x in G'.*

Proof: The proof is an induction on the number of back-edges removed from G. Initially, suppose that G has only a single back-edge (u, v), a is a control ancestor of x in G, but a is no longer a control ancestor of x when G' is created from G by removing (u, v). By the definition of control ancestor, there exists some acyclic control path π_x in G from $START$ to x that contains a. Because a is not a control ancestor of x in G', we must have $(u, v) \in \pi_x$. By the definition of a PPG and back-edges, there must be an acyclic control path π_v from $START$ to v such that $u \notin \pi_v$.

Case 1: $a \in \pi_v$. Since π_x contains a subpath from v to x, $a \in \pi_v$ implies that there is a control path from a to x that does not contain (u, v). Consequently, a remains a control ancestor of x after the removal of (u, v), a contradiction.

Case 2: $a \notin \pi_v$. Because (u, v) is a back-edge, v is a control ancestor of u, and consequently v is the entry point of some cycle containing (u, v). Because $a \notin \pi_v$, there must be a node on π_x that is another entry point to that cycle. Consequently, (u, v) belongs to a multiple-entry cycle, a contradiction.

If G has k back-edges, we instead consider $\tilde{G}$, which is constructed from G by removing back-edges such that the set of control ancestors of each node remains unchanged, and the removal of any additional back-edge would cause some node to loose a control ancestor. We then apply the above argument to $\tilde{G}$. □

Lemma 9. *Let $G = (N, E)$ be a PPG, and assume that all cycles in G are single-entry, all synchronization edges are control-independent, and none is a back-edge. Then G will not deadlock.*

Proof: Let $\tilde{G}$ be the graph that it obtained from G by removing all back-edges from G. By lemma 8 every node in N has the same set of control ancestors in G and in $\tilde{G}$. Since all back-edges are control edges, G and $\tilde{G}$ have the same synchronization edges. By lemma 7 $\tilde{G}$ never deadlocks.

Assume for contradiction G deadlocks, and let $v \in N$ be a deadlocking node in G with deadlocking instance v_i such that no predecessor of v in $\tilde{G}$ is a deadlocking node in G. By lemma 6 there exists a synchronization edge (u, v, f) that is relevant for v_i and which is causing v_i to deadlock. If v is not part of a cycle in G, then the arguments of lemma 7 hold. If v is part of a cycle, then since u is a predecessor of v in $\tilde{G}$, no instance u_j of u that is created deadlocks. Consequently, since (u, v, f) is relevant, there must be some instance u_j of u such that $f(H(I_{u_j}), H(I_{v_i})) = true$, but u_j is not created. It follows from the definition of control-independent synchronization edges that $nodeprefix(H(I_{u_j}), a) = nodeprefix(H(I_{v_i}), a)$ for all nodes a that are control

ancestors of both u_j and v_i. Therefore, if v_i is created, so is u_j, and consequently v_i cannot deadlock. □

Theorem 10. *If all cycles are single-entry, the presence of a synchronization edge that is a potential back-edge or is not a control-independent edge is necessary for deadlock to occur.*

Proof: Follows from lemma 11. □

Let C be an multiple-entry cycle with entry points $x_1, x_2, \ldots, x_k$. We define C_{x_i} to be the single-entry cycle that is obtained by deleting all edges $(y, x_j), j \neq i$, such that $y \notin C$. We say that C_{x_i} is a *single-entry cycle with entry point* x_i. Edge (v, x_i) is a *potential* back-edge if (v, x_i) is a back-edge of C_{x_i}. A back-edge is also a potential back-edge.

We conjecture that lemma 11 and theorem 12 hold, but we have not yet formalized the proof.

Lemma 11. *[Conjecture]. Suppose G contains multiple-entry cycles. If all synchronization edges are control-independent and none is a potential back-edge, then G cannot deadlock.*

Theorem 12. *[Conjecture]. The presence of a synchronization edge that is a potential back-edge or is not a control-independent edge is necessary for deadlock to occur.*

We now show that determining if there exists an execution sequence that deadlocks in a PPG with acyclic control edges is NP-hard. The transformation is from *3-SAT* [17]: Given a Boolean expression $E = (a1_i \vee a1_j \vee a1_k) \wedge (a2_i \vee a2_j \vee a2_k) \wedge \ldots \wedge (am_i \vee am_j \vee am_k)$, with each clause $c_p = (ap_i \vee ap_j \vee ap_k)$ containing 3 literals, $1 \leq p \leq m$. Assume all the literals that occur in E are either positive or negative forms of variables $a_1, a_2, \ldots, a_n$. Is there a truth assignment to the variables (predicates) such that E evaluates to *true*?

Theorem 13. *Determining if there exists an execution sequence that contains a deadlocking cycle is NP-hard for a PPG with an acyclic control dependence graph.*

Proof: To prove that the problem is NP-hard, let E be an instance of *3-SAT* with n variables, $a_1, a_2, \ldots, a_n$, and m clauses, $c_1, c_2, \ldots, c_m$. We assume that there is some occurrence of both the *true* and *false* literals of every variable in E, since otherwise we could just set the truth assignment of the variable equal to the value that makes all of its literals true, and eliminate all clauses containing that literal.

We construct a PPG G with a *START* node, n (variable) predicate nodes, $2n + 1$ *MGOTO* nodes, and m (clause) *COMPUTE* nodes. We first describe the control edges of G. *START* has an *MGOTO* node as its only child. That *MGOTO* node has the n predicate nodes as its children. Each predicate node corresponds to one of the variables that occur in E. If A_i is the predicate node

corresponding to the variable a_i, then the *true* branch of A_i has an $MGOTO$ node as its child, and the *false* branch of A_i has a second $MGOTO$ node as its child. Let c_r be a clause in E with corresponding $COMPUTE$ node C_r in G, and suppose that $C_r = (a_i \vee \bar{a}_j \vee a_k)$. Then there are control edges from the $MGOTO$ children of the *true* edges of A_i and A_k and from the *false* edge of A_j to C_r. In general, if a_i occurs in c_j in its *true* (*false*) form, then there is an edge from the $MGOTO$ descendant of the *true* (*false*) branch of A_i to C_j in G.

The synchronization edges are directed from C_i to C_{i+1} for $1 \leq i < m$, and from C_m to C_1. The synchronization condition, f, for the synchronization edges is ALL i.e. all instances of the source node must complete execution before any instance of the destination node can start execution. For each $COMPUTE$ node, the *wait* operations for all incoming synchronization edges must be completed before the *signal* operation for any outgoing synchronization edges can be performed. So a $COMPUTE$ node will send its synchronization signal to its neighbor only after having received a similar signal from its other neighbor.

If all $COMPUTE$ nodes have at least one instantiation, then the execution sequence for G will deadlock. If one of the $COMPUTE$ nodes C_i is not executed at all, then no instance of C_{i+1} needs to wait for synchronization messages from C_i, and so the instances of C_{i+1} will complete execution and send their synchronization messages to instances of C_{i+2}, which will then send messages to instances of C_{i+3}, and so on.

If there is a truth assignment to the variables of E so that E is satisfied, then for that truth assignment each $COMPUTE$ node will be executed at least once, and deadlock will occur. Conversely, if G deadlocks, then we can obtain a truth assignment to the variables of E such that E is satisfied. $\Box$

For each clause node there are three node disjoint paths from the $MGOTO$ child of $START$ to that clause node. Therefore, the PPG used in the above reduction does not satisfy the pred-anc cond[4]. The effect of the pred-anc cond is that in a PPG with acyclic control edges, a node is executed only zero or one times. In section 3, Figures 1 and 3 are examples of PPGs that satisfy the anc-prec cond, while Figure 2 is an example of a PPG that does not satisfy the anc-prec cond. As we show in section 6, the pred-anc cond is a necessary, but not sufficient, condition for a PPG to be serializable without node duplication or the introduction of guard variables. Furthermore, if the predicate-ancestor and no-post-dominator conditions hold, then determining whether or not a PPG can be serialized without node duplication can be done in polynomial time.

An obvious question is: if the pred-anc cond holds, can deadlock detection also be done in polynomial time? One observation is that if the PPG is found to be serializable, then we can assert that it will not deadlock. So, the reduced question is: if the pred-anc condition holds and the PPG is known to be non-serializable, can deadlock detection also be done in polynomial time? Since we don't have the answer to that question, a possibly easier but related question is the following: Given a set of PPG nodes, none of which is a control ancestor of the other, and assuming that the pred-anc cond holds, can we determine in

[4] But it satisfies the no post-dominance rule.

polynomial time if there exists an execution sequence such that all the nodes in the set are executed? We are also unable to answer this question, but we can give a necessary and sufficient characterization for the existence of an execution sequence in which all the nodes in the set are executed. This characterization implies a fast algorithm for detecting "bogus" edges in an acyclic PPG that satisfies the pred-anc cond.

Lemma 14. *Let G be an acyclic PPG and assume that the pred-anc cond holds. Let S be a set of nodes in G such that none is a control ancestor of the other. Then there exists an execution sequence of G in which all the nodes in S are executed if and only if there is a rooted spanning tree T (not necessarily spanning all the nodes of G) consisting of only control edges of G such that all nodes in T with multiple out-edges are $MGOTO$ nodes and all nodes in S are leaves in T.*

Proof: If such a T exists, then since the root of T is reachable from $START$ and since all out-edges of an $MGOTO$ nodes are always taken, then there exists an execution sequence for which all the nodes of S are executed. Conversely, suppose that there exists an execution sequence such that all the nodes of S are executed. Let G' consist of only the nodes and edges in G that are executed in that execution sequence. Because the pred-anc cond implies that every node is executed zero or one time, if two nodes $X, Y \in S$ have a predicate least common control ancestor P_1 in G', then since P_1 is executed at most once, at most one of X and Y will be executed whenever P_1 is executed. If X and Y have two predicate least common ancestors, P_1 and P_2, then there must be two node disjoint paths to each of X and Y, one containing P_1 and the other containing P_2. Therefore, by the pred-anc cond, the least common ancestor of P_1 and P_2 must be a predicate P_3, and at most one of P_1 and P_2, and consequently X and Y, will be executed whenever P_3 is executed. It follows from induction that the least common control ancestor in G' of any pair of nodes in S must be an $MGOTO$ node. A similar argument shows that any pair of these $MGOTO$ either lie in a path in G' or have another $MGOTO$ node as their least common ancestor. Finally, the configuration of the $MGOTO$ nodes and the nodes in S must be a tree, since otherwise some $MGOTO$ node would violate the pred-anc cond. □

Definition 15. A synchronization edge (a, b, f) is said to be "*bogus*" if there is no execution sequence of the PPG in which it is executed i.e. there are no possible instantiations I_a and I_b of nodes a and b that satisfy $f(H(I_a), H(I_b)) = true$ and that can be obtained from the same execution of the PPG.

Corollary 16. *If an acyclic PPG satisfies the pred-anc cond, then all bogus synchronization edges can be detected in $O(|E_{sync}| \times |E_{cont}|)$ time, where $|E_{sync}|$ is the number of synchronization edges and $|E_{cont}|$ is the number of control edges in G.*

Proof: Lemma 14 implies that a synchronization edge (u, v, f) is bogus if and only if there is no $MGOTO$ node which is the least common ancestor of both

u and v in G. Consequently, a simple backtracking search from each endpoint of the synchronization edge being tested is sufficient to determine if the two endpoints have an *MGOTO* least common ancestor. □

6 Serializability of PPGs

After a PPG has been used to optimize and possibly parallelize code, the optimized code frequently must be mapped into sequential target code for the individual processor(s). An equivalent problem can arise when trying automatically to merge different versions of a program which have been updated in parallel by several people [22, 23].

Serialization customarily involves transforming the PPG to a CFG, from which the sequential code is then derived. Whenever possible, the CFG is constructed without duplicating nodes or inserting Boolean guard variables. Node duplication is undesirable because of potential space requirements which can create cache and page faults, and the introduction of guard variables is undesirable because of the additional time and space needed to store and test such variables.

6.1 The model.

In this section, we analyze a restricted model of a PPG, as defined in [16, 38, 37]. We allow only the *MGOTO* node to have multiple control in-edges. This is not a restriction on the model, since if a *PREDICATE* or *COMPUTE* node a has multiple control in-edges, we can always direct the in-edges into a new *MGOTO* node f, which is then made the (unique) parent of a. We also assume that all *COMPUTE* nodes are leaves in the control subgraph of the PPG, and that all predicates have precisely two out-going control edges, one labelled *true* and the other labelled *false*.

Finally, we assume that pred-anc condition holds. This is indeed a restriction, since if the pred-anc cond does not hold, then any CFG constructed from a PPG that violates the pred-anc cond provably requires node duplication, as is illustrated in figure 4. In the example of figure 4 the least common ancestor of two paths terminating at $s3$ is *MGOTO* node $f1$. Therefore, there is an execution sequence in which $s3$ is executed twice, and any sequential representation will have two copies of $s3$.

Since a CFG has no parallelism, construction of a CFG from a PPG involves the elimination of *MGOTO* nodes and the merging of the subgraphs of their children. For the remainder of this section, we assume as input a PPG that satisfies the pred-anc cond and the node restrictions discussed above.

Even if the pred-anc cond is satisfied, there are PPGs for which node duplication or guard variable insertion is unavoidable [37].

The Control Dependence Graph (CDG) is the subgraph of the PPG that represents only the control dependences (or control edges). The CDG encapsulates the difficulty of translating the PPG into a CFG. The result presented in

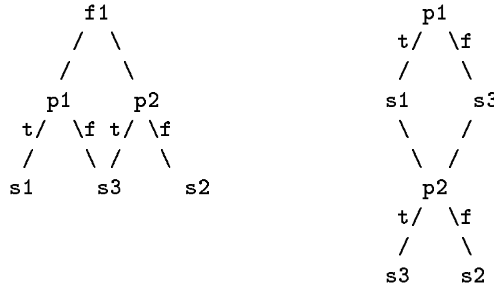

Fig. 4. A PPG subgraph and equivalent sequential CFG subgraph with duplication

this section also assumes that the CDG satisfies the no post-dominator condition defined in section 4.

In order to determine when a PPG can be serialized, we need some notion of equivalence between a CDG and a CFG. One approached, taken by Ball and Horwitz [6], is to say that a CFG G' is a *corresponding* CFG of CDG G if and only if the CDG that is constructed from G' is isomorphic to G, with the isomorphism respecting node and edge labels. We have chosen a definition that does not depend on isomorphism [37, 36].

Given two graphs G and G', where G is an acyclic CDG and G' is a CFG, we say that G' is *semantically equivalent* to G if:

1. All *PREDICATE* and *COMPUTE* nodes in G' are copies of nodes in G, and every *PREDICATE* or *COMPUTE* node in G has at least one copy in G'.
2. For every truth assignment to the predicates, the *PREDICATE* and *COMPUTE* nodes that are executed in G' are copies of exactly those *PREDICATE* and *COMPUTE* nodes executed in G. If *PREDICATE* or *COMPUTE* node a is executed $c(a)$ times in G under some truth assignment, then the number of times a copy of a is executed in G' is equal to $c(a)$[5].

Because a CDG G has no post-dominance, one can prove that the following condition must hold in any CFG G' that corresponds to an acyclic CDG G that satisfies the pred-anc cond [36].

3. Suppose a and b are *PREDICATE* or *COMPUTE* nodes in G. Then if a precedes b in G, then no copy of b precedes a copy of a in G'.

The definition of semantic equivalence is extended to cyclic graphs with reducible loops by first applying it to the acyclic graphs in which the loops are replaced by "supernodes" and then applying the definition recursively to the acyclic subgraphs within corresponding supernodes.

[5] If the PPG satisfies the pred-anc cond, then $c(a) \leq 1$.

6.2 Algorithms for constructing a CFG from a PDG

There are two algorithms for constructing a CFG from a PDG, each of which uses a somewhat different model for the PDG. Both assume that all loops in the PDG are reducible and that the no-post-dominator condition holds; both have a running time of $O(ne)$, where n is the number of nodes and e is the number of edges in the PDG. The complexity involves the analysis of the control dependences.

Constructing a CFG and reconstituting a CDG. The algorithm presented in [6] determines orderings that must exist in any CFG G' that "corresponds" to the CDG G[6]. If the algorithm fails to construct a CFG G', or if it constructs a CFG that does not correspond to G, i.e. for which there is not a CDG that is isomorphic to G, then the algorithm fails. Although the algorithm does not explicitly deal with data dependences, it can be modified to do so.

The difficulty with the above approach is that when the algorithm fails, it does not provide any information as to why it fails.

Algorithm Sequentialize. Algorithm Sequentialize [37, 36] is based on analysis that states that a CFG G' can be constructed from a PDG G without node duplication if and only if there is no "forbidden subgraph". The intuition behind the forbidden subgraph is that duplication is not required if and only if there is no set of "forced" execution sequences which would create a cycle in the acyclic portion of the graph. A forbidden subgraph implies the existence of such a cycle. A cycle can also exist if data dependences contradict an ordering implied by the control dependences. When duplication is required, the algorithm can be used as the basis of a heuristic that duplicates subgraphs or adds guard variables.

7 Related Work

The idea of extending PDGs to PPGs was introduced by Ferrante and Mace in [14], extended by Simons, Alpern, and Ferrante in [38], and further extended by Sarkar in [34]. The definition of PPGs used in this paper is similar to the definition from [34]; the only difference is that the definition in [34] used *mgoto edges* instead of *mgoto nodes*. The PPG definition used in [14, 38] imposed several restrictions: a) the PPGs could not contain loop-carried data dependences (synchronizations), b) only reducible loops were considered, c) PPGs had to satisfy the no-post-dominator rule, and d) PPGs had to satisfy the pred-anc rule. The PPGs defined in this paper have none of these restrictions. Restrictions a) and b) limit their PPGs from being able to represent all PDGs that can be derived from sequential programs; restriction a) is a serious limitation in practice, given the important role played by loop-carried data dependences in representing loop parallelism [40]. Restriction c) limits their PPGs from being

[6] The model presented in [6] differs somewhat from that of [37, 36]. However, the models are essentially equivalent.

able to represent CFGs. Restriction d) limits their PPGs from representing thread-based parallel programs in their full generality e.g. the PPG from Figure 2 cannot be expressed in their PPG model because it is possible for node S4 to be executed twice.

The Hierarchical Task Graph (HTG) proposed by Girkar and Polychronopoulos [18] is another variant of the PPG, applicable only to structured programs that can be represented by the hierarchy defined in the HTG.

One of the central issues in defining PDGs and PPGs is in defining their parallel execution semantics. The semantics of PDGs has been examined in past work by Selke, by Cartwright and Felleisen, and by Sarkar. In [35], Selke presented an operational semantics for PDGs based on graph rewriting. In [10], Cartwright and Felleisen presented a denotational semantics for PDGs, and showed how it could be used to generate an equivalent functional, dataflow program. Both those approaches assumed a restricted programming language (the language W), and presented a value-oriented semantics for PDGs. The value-oriented nature of the semantics made it inconvenient to model store-oriented operations that are found in real programming languages, like an update of an array element or of a pointer-dereferenced location. Also, the control structures in the language W were restricted to `if-then-else` and `while-do`; the semantics did not cover programs with arbitrary control flow. In contrast, the semantics defined by Sarkar [34] is applicable to PDGs with arbitrary (unstructured) control flow and arbitrary data read and write accesses, and more generally to PPGs as defined in this paper.

The serialization problem has been addressed in some prior work. Generating a CFG from the CDG's of well-structured programs is reported in [22, 23]; such CDG's are trees and do not require any duplication. The IBM PTRAN system [3, 9, 33] generates CFGs from the CDGs of unstructured (i.e. not necessarily trees) FORTRAN programs. The work presented in [14, 16] uses assumptions about the structure of CFG's derived from sequential programs, thereby limiting its applicability. The results presented in [16] consider the general problem in an informal manner by determining the order restrictions that had to hold for region node children of a region node.

8 Conclusions and Future Work

In this paper, we have presented definitions and classifications of Parallel Program Graphs (PPGs), a general parallel program representation that also encompasses PDGs and CFGs. We believe that the PPG has the right level of semantic information to match the programmer's way of thinking about parallel programs, and that it also provides a suitable execution model for efficient implementation on real architectures. The algorithms presented for deadlock detection and serialization for special classes of PPGs can be used to relieve the programmer from performing those tasks manually. The characterization results can be viewed as also identifying PPGs that are easier for programmers to work with e.g. PPGs for which deadlock detection and/or serializability are hard to

perform are quite likely to be PPGs that are hard for programmers to understand and that lead to programming errors.

We see several possible directions for future work based on the results presented in this paper:

- *Extend sequential program analysis techniques to the PPG*
  All program analysis techniques currently used in compilers (e.g., constant propagation, induction variable analysis, computation of Static Single Assignment form, etc.) operate on a sequential control flow graph representation of the program. There has been some recent work in the area of program analysis in the presence of structured parallel language constructs [28, 39]. However, just as the control flow graph is the representation of choice (due to its simplicity and generality) for analyzing sequential programs, it would be desirable to use the PPG representation as a simple and general representation for parallel programs.
  In particular, we are interested in extending to PPGs the notions of dominators and control dependence that have been defined for CFGs. These extensions should provide us with a way of automatically transforming a given PPG into a form that satisfies the no-post-dominator condition, so that assuming the no-post-dominator condition will not restrict the applicability of the results obtained from that assumption.
- *Investigate the complexity of deadlock detection for acyclic PPGs that satisfy the pred-anc and no-post-dominator conditions*
  In section 5, we showed that the problem of detecting deadlock in PPGs with acyclic control edges is NP-hard. Moreover, for PPGs with acyclic control edges that satisfy the pred-anc and no-post-dominator conditions, section 6 provides a polynomial-time algorithm for determining whether or not the PPG is serializable. We have observed that serializable PPGs cannot deadlock. Therefore, the remaining open issue is: what is the complexity of deadlock detection for PPGs that satisfy the pred-anc and no-post-dominator conditions but are non-serializable? We are interested in resolving this open issue by either obtaining a polynomial-time deadlock detection algorithm for such PPGs or by proving that the problem is NP-hard.
- *Develop a transformation framework for PPGs*
  Current parallelizing compilers optimize programs for parallel architectures by performing transformations on perfect loop nest, on CFGs, and on PPGs. It would be interesting to extend existing algorithms and design new algorithms in a transformation framework for PPGs. In this paper, we discussed serialization and the elimination of bogus edges, which can also be viewed as transformations on PPGs. In the future, we would like to perform other transformations to match the performance characteristics and code generation requirements of different parallel architectures.
- *Develop a common compilation and execution environment for different parallel programming languages*
  The PPG can be used as the basis for a common compilation and execution environment for parallel programs written in different languages.

The scheduling system defined in [34] can be developed into a PPG-based interpreter, debugger, or runtime system for parallel programs, that provides feedback to the user about the parallel program's execution (e.g. parallelism profiles, detection of read-write or write-write hazards, detection of deadlock, etc.). Currently, such programming tools are being developed separately for different languages, with different (ad hoc) assumptions being made about their parallel execution semantics. The PPG representation could be useful in leading to a common environment for different parallel programming languages.

- *Study the limitations of the PPG representation*
  Though we showed in this paper how PPGs are more general than PDGs and CFGs, we are aware that there are certain parallelism constructs that do not map directly to PPGs. Examples of such constructs include general post-wait synchronizations in which the synchronization condition depends on data values (not just on execution histories) and single-program-multiple-data (SPMD) execution in which the number of times a region of code is executed can depend on a run-time parameter such as the number of processors. It would be interesting to see how such constructs can be represented by PPGs without losing too much semantic information. We conjecture that parallel programming constructs that are hard to represent by PPGs are also hard for programmers to understand. A study of the limitations of PPGs will shed more light on this conjecture.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 19(6):13–24, 1970.
3. Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Proceedings of the ACM 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, 5(5) pages 617-640.
4. Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–592, October 1987.
5. Arvind, M.L. Dertouzos, R.S. Nikhil, and G.M. Papadopoulos. Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language. Technical report, MIT Lab for Computer Science, March 1988. Computation Structures Group Memo 285.
6. Thomas Ball and Susan Horwitz. Constructing Control Flow from Data Dependence. Technical report, University of Wisconsin-Madison, 1992. TR No. 1091.
7. William Baxter and J. R. Bauer, III. The Program Dependence Graph in Vectorization. *Sixteenth ACM Principles of Programming Languages Symposium*, pages 1 – 11., January 11-13 1989. Austin, Texas.
8. Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*,

21(7):162–175, July 1986.

9. Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic Generation of Nested, Fork-Join Parallelism. *Journal of Supercomputing*, 2(3):71–88, July 1989.
10. Robert Cartwright and Mathias Felleisen. The Semantics of Program Dependence. *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, June 1989.
11. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences Using Control Dependence in PTRAN. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989. In Languages and Compilers for Parallel Computing, edited by D. Gelernter, A. Nicolau, and D. Padua, MIT Press, 1990 (pages 186-212).
12. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact Representations for Control Dependence. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, pages 337–351, June 1990.
13. Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic Generation of DAG Parallelism. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, 24(7):54–68, June 1989.
14. J. Ferrante and M. E. Mace. On Linearizing Parallel Code. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, pages 179–189, January 1985.
15. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
16. Jeanne Ferrante, Mary Mace, and Barbara Simons. Generating Sequential Code From Parallel Code. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 582–592, July 1988.
17. Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.
18. Miland Girkar and Constantine Polychronopoulos. The HTG: An Intermediate Representation for Programs Based on Control and Data Dependences. Technical report, Center for Supercomputing Res. and Dev.-University of Illinois, May 1991. CSRD Rpt. No.1046.
19. Rajiv Gupta and Mary Lou Soffa. Region Scheduling. *Proc. of the Second International Conferenece on Supercomputing*, 3:141–148, May 1987.
20. P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software engineering*, SE-1(2):199–206, June 1975.
21. Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., 1977.
22. Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 133–145, January 1987.
23. Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 146–157, January 1987.
24. Paul Hudak and Philip Wadler et al. Report on the Functional Programming Language Haskell. Technical report, Yale University, 1988. Research Report YALEU/DCS/RR-666.

25. Peter Ladkin and Barbara Simons. Compile-Time Analysis of Communicating Processes. *Proc. of the ACM 1992 International Conference on Supercomputing*, pages 248–259, July 1992.
26. L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.
27. Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array Data-Flow Analysis and its Use in Array Privatization. *Conf. Rec. Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.
28. Samuel Midkiff, David Padua, and Ron Cytron. Compiling Programs with User Parallelism. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.
29. PCF. Parallel Computing Forum, Final Report. Technical report, Kuck and Associates, Incorporated, 1990. In preparation.
30. W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, California*, pages 140–151, June 1992.
31. Vivek Sarkar. Determining Average Program Execution Times and their Variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.
32. Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.
33. Vivek Sarkar. The PTRAN Parallel Programming System. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.
34. Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). Technical Report YALEU/DCS/RR-915, Yale University, Department of Computer Science, August 1992. Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing, Yale University, August 3-5, 1992.
35. Rebecca Parsons Selke. A Rewriting Semantics for Program Dependence Graphs. *Sixteenth ACM Principles of Programming Languages Symposium*, January 11-13 1989. Austin, Texas.
36. B. Simons. Constructing a Constructing a Control Flow Graph for Parallel Code. Technical report. To appear.
37. B. Simons and J. Ferrante. An Efficient Algorithm for Constructing a Control Flow Graph for Parallel Code. Technical report, IBM, February 1993. Technical Report TR 03.465-1.
38. Barbara Simons, David Alpern, and Jeanne Ferrante. A Foundation for Sequentializing Parallel Code. *Proceedings of the ACM 1990 Symposium on Parallel Algorithms and Architecture*, July 1990.
39. Harini Srinivasan and Michael Wolfe. Analyzing Programs with Explicit Parallelism. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991. To be published by Springer-Verlag.
40. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

This article was processed using the LaTeX macro package with LLNCS style